Implementing the Game of Life with Concurrent Methods

Tom Jager, Computer Science, tj15299@my.bristol.ac.uk
William Rollason, Computer Science, wr15339@my.bristol.ac.uk

Functionality and Design

getAlive:

In order to implement the Game of Life rules, we created a function called getAlive which takes 3 parameters, world, x and y. World is a uchar array which holds the cells being processed as bits where 1 indicates a live cell and 0 a dead cell. getAlive determines whether a specific cell (denoted by the coordinates x,y) is alive in the next iteration of the game. As the board is wrapped horizontally, the first cell off of the right edge of the board is given as the first cell on the left and vice versa. Furthermore, as each cell is stored as a bit and not as an individual byte, we used bit manipulation to determine how many neighbours of the cell were alive. getAlive returns 1 if cell x,y is alive in the next iteration and 0 if it is not.

Distributor:

The distributor function can be split into four sections. The reading functionality, the processing of data, the pause functionality and the export of the processed image.

Reading Functionality

When the program is run the board waits for a signal from the buttonListener process indicating that SW1 has been pressed. This then initiates the read in of the image where each pixel value is stored as a single bit in an array of bytes. During the read in process the 1st green LED is turned on. Once the image has been stored, the board signals the timer to start.

Data Processing

Once the image is read we enter a while loop that runs until the distributor is told to export. Here the runGame function is run to distribute the data between workers and store the processed data from the workers back in the board array.

runGame takes two parameters, board and workerVal. Board is the entire game board at the moment in time where each cell is denoted by a 1 or 0 bit. workerVal is a channel array which contains the channel ends that communicate to each worker process. Inside runGame we partition the board into equally sized sections and send them to each worker. Each worker receives the line above the data it will actually be working on, then the lines it will processing and then the line below the data it will be processing. First we send on the first worker's channel the very bottom line of the entire board. This ensures that the board will be wrapped vertically. We then pass it the remaining lines it needs. We repeat this for all the workers, passing the last worker the top line of the board to again ensure vertical wrapping.

The board array is updated with the values of next round once the workers have finished processing their data. The workers send the updated values back in order from the first process to the last to ensure the data is written to the correct locations in the board. Each worker also sends a count of the number of live cells in their area which is totaled. runGame then returns the total number of live cells.

The live cell total is stored in a variable called count. After the runGame function has finished, the round counter is incremented and the adjacent green LED is turned on or off depending on what it was before.

Pausing Functionality

If the board receives a signal from the orientation sensor to indicate a tilt the processing is paused, red LED turned on and another signal sent to the timer. During a pause the distributor receives the time elapsed for processing from the timer and also the number of alive cells in the current round. It prints a status report that displays the current round, the number of live cells and time taken to process the image thus far. If another signal is received for the board being returned to flat the pause loop breaks and processing continues.

Image Export

When the sw2 button is pressed the distributor receives a signal from the button port and the processing loop is broken. This turns on the blue LED and begins an export of the current board. Each uchar in board stores 8 pixels as bits so each bit must be converted back into a uchar representing either 0 or 255 and then transmitted to the DataOutStream thread which writes the pixel values into a PGM image file.

timeCalc

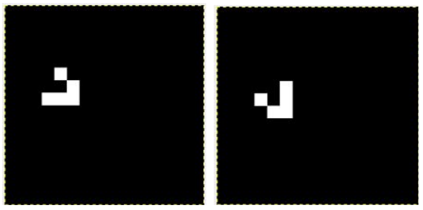
timeCalc is a process that calculated how much time has elapsed between signals.The timer is started when a signal is received from the distributor and every time a second passes the variable timeCount is incremented by 1000. When the timer receives a request signal from the distributor it calculates the current time by subtracting the time at the last second from the current time and then adding this to the number of seconds that have passed. This value is then sent back to the distributor to be used in the status report. A separate timeCount variable must be used to store the number of milliseconds passed as the timer resets every 42 seconds.

Worker

The worker processes are the parallel running processes that calculate the values of the cells for the next round of the Game of Life. Each worker take a channel end that communicates to the distributor. Each worker contains two arrays. The first array, board, contains the current status of the cells that are being processed by that worker, as well as the lines above and below them which are needed to calculate their status in the next iteration. The second array, result is initialised as a blank array but will hold the processed values of each cell. The worker then enters a while loop where it receives the current cell values it requires from the distributor, calls getAlive on each relevant cell and increases a counter by the result of getAlive. To store the cell as a single bit in an array of bytes, we left shift the appropriate byte by 1 and OR the byte with the value returned by getAlive. After all the cells have been calculated we transmit the result array back to the distributor using the channel and then send over the value of the alive cells counter.

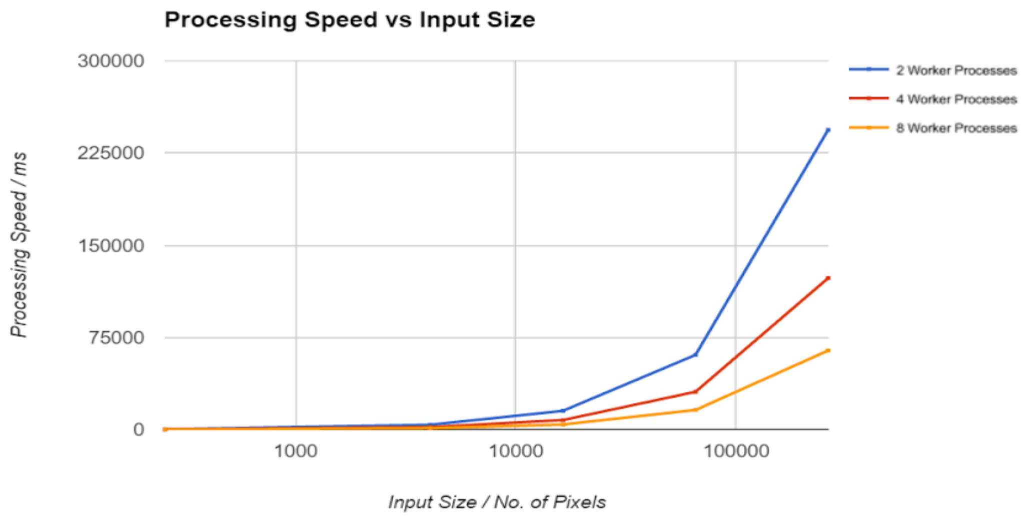
Tests and Experiments

The first test run on the board was 2 iterations of the 16x16 glider test image provided. This proved that our initial basic implementation worked.



After improving our program we ran several timed test on all of the different sized images provided for 100 rounds, trying all combinations of 2, 4 and 8 workers on each image size. The timing ignored both read in and read out time only returning the time taken to actually process the image. We collated the results into a table and created a graph to show the log of image size against processing time.

	Processing Speed / ms		
Input Size	2 Workers	4 Workers	8 Workers
256	340	224	166
4096	3940	2041	1134
16384	15266	7785	4196
65536	60764	30763	15969
262144	243840	123223	64336



We have also tried using our own images created on the board up to the size of 896 x 896 anything larger than this and the board runs out of memory with our current implementation.

Critical Analysis

Initially we implemented the program with only two workers and storing each pixel as a uchar in a 2D array. This allowed us to write an efficient program for processing a 16 x 16 image quickly but the program would crash with anything larger. To overcome this we decided to use bit packing to be able to store the image in a uchar array an eighth the size of before. This allowed us to read in much larger images up to the maximum test image size of 512 x 512 as each pixel required only one bit of memory instead of a byte. Here we decided to increase the number of worker threads processing the data to speed up computation. Whilst 4 workers on one thread was acceptable, increasing the number of workers to 8 created errors. To fix this we had to allocate half the worker processes to tile[0] and the other half to tile[1]. This allowed us to use memory on both cores. Having 8 workers drastically increased the processing speed, almost quartering computation time with the larger images compared to 2 workers. At the 512 x 512 image size with 8 workers the board was taking on average 643ms on each round. The biggest bottleneck in our system was input and output of the image which took a lot more time than the actual processing. The time for input was reduced when we started writing our own images on the board but exporting the result was still slow. To fix this we could potentially parallelise the in and out stream processes, dividing them amongst worker processes.

Further improvements in the code could be made by using geometric parallelisation rather than farming. When using farming, processing time is wasted dividing the data and combining it at the beginning and end of each round. With geometric parallelisation you could would only need to transmit the lines above and below processing areas between workers which would decrease processing time.